Artificial Intelligence

Heuristic search

You can use DFS (Depth-first search) or BFS (Breadth-first search) to explore a graph, but it may require $O(b^m)$ node visits, but if you want to search more efficiently, it might be better to check out the promising leads first. With a heuristic search, at each step of the way, you use an evaluation function to choose which subtree to explore. It is important to remember that a heuristic search is best at finding a solution that is "good enough" rather than the perfect solution. Some problems are so big that finding the perfect solution is too hard. To do a heuristic search you need an evaluation function that lets you rank your options.

Heuristic Search takes advantage of the fact that most problem spaces provide, at relatively small computational cost, some information that distinguishes among the states in terms of their likelihood of leading to a goal state. This information is called a "heuristic evaluation function".

Informed search strategy uses problem specific knowledge beyond the definition of the problem itself. It can find solutions more efficiently than an uninformed strategy. A heuristic is a technique that improves the efficiency of a search process possibly by sacrificing claims of incompleteness.There are some good general purpose heuristics that are useful in a wide variety of problem domains. In addition it is posssible to construct special purpose heuristics that exploit specific knowledge to solve particular problems.

One example of a geenral purpose heuristics that is used in a wide variety of problems is the nearest neighbour heuristics which works by selecting the locally superior alternative at each step. Applying it to TSP we produce the following procedure.

Nearest neighbor heuristic:

- Select a starting city.
- To select the next city, look at all cities not yet visited and select the one closest to the current city. Go to it next.
- Repeat step 2 until all cities have been visited.

If the generation of possible solutions is done systematically, then this procedure will find a solution eventually, if one exists. Unfortunately if the problem space is very large, this may take very long time. The generate and test algorithm is a depth first search procedure since complete solutions must be generated before they can be tested. Generate and test operate by generating solutions randomly, but then there is no guarantee that a solution will ever be found. The most straight forward way to implement systematic generate and test is as a depth first search tree with backtracking.

Prepared By:

Er. Gurjot Singh Sodhi
Asst. Prof., Dept. of CSE
SUSCET, Tangori.

In heuristic search a node is selected for expansion based on an evaluation function, f(n). Traditionally the node with the lowest evaluation is selected for expansion, because the evaluation measures the distance to the goal. All that we do is choose the node that appears to be best according to the evaluation function. If the evaluation function is exactly accurate then this will indeed be the best node.

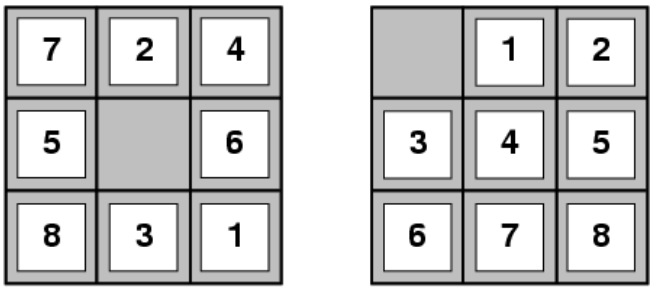
A key component of all these heuristic search algorithms is a **heuristic function** denoted by h(n):

h(n)= estimated cost of cheapest path from node n to the goal node.

Heuristic functions are the most common form in which additional knowledge of a problem is imparted to the search algorithm. If n is the goal node then h(n)=0.

Heuristic Functions

8-puzzle was one of the earliest heuristic search problems. The object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration.



Above shown is a particular instance of 8-puzzle. The solution is 26 steps long. If we use a heuristic function number of steps needed to reach the goal can be reduced. The most commonly used are:

1. h_1 = Number of tiles out of place

In the figure all of the 8 tiles are out of position, so the start state would have $h_1 = 8$. h_1 is an admissible heuristics, because it is clear that any tile out of place must be moved at least once.

2. h_2 = the sum of the distances of the tiles from their goal positions.

Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is sometimes called the city block distance or Manhattan distance. h_2 is an admissible heuristics, because all any move can do is move one tile one step closer to the goal. Tiles 1 to 8 in the start state give a Manhattan distance of

$$h_2 = 3+1+2+2+2+3+3+2 = 18$$

Prepared By:

Er. Gurjot Singh Sodhi
Asst. Prof., Dept. of CSE
SUSCET, Tangori.

As we would hope neither of these overestimates the true solution cost which is 26. An admissible heuristic function never overestimates the distance to the goal. The function h=0 is the least useful admissible function. Given 2 admissible heuristic functions (h1 and h2), h1 **dominates** h2 if $h1(n) \geq h2(n)$ for any node n. The perfect heuristic function is dominant over all other admissible heuristic functions. Dominant admissible heuristic functions are better.

The effect of heuristic accuracy on performance

One way to characterize the quality of a heuristics is the **effective branching factor** b*. If the total number of nodes generated by A* for a particular problem is N, and the solution depth is d, then b* is the branching factor that a uniform tree of depth d would have to have in order to contain n+1 nodes. Thus

$$N+1 = 1+b^*+(b^*)^2+\ldots\ldots\ldots\ldots+(b^*)^d.$$

The effective branching factor can vary across problem instances, but usually it is fairly constant for sufficiently hard problems. A well designed heuristic would have a value of b* close to 1, allowing fairly large problems to be solved.

Inventing admissible heuristic functions

The problem of finding an admissible heuristic with a low branching factor for common search tasks has been extensively researched in the artificial intelligence community. Common techniques used are:

- **Relaxed Problems**

Remove constraints from the original problem to generate a "relaxed problem". A problem with fewer restrictions on actions is called a **relaxed problem**. Cost of optimal solution to relaxed problem is admissible heuristic for original problem. Because a solution to the original problem also solves the relaxed problem (at a cost $\geq$ relaxed solution cost). For example, Manhattan distance is a relaxed version of the n-puzzle problem, because we assume we can move each tile to its position independently of moving the other tiles.

If the problem definition is written down in a formal language, it is possible to construct relaxed problems automatically. For example if the 8-puzzle actions are described as:

A tile can move from square A to square B if --- A is horizontally or vertically adjacent to B and B is blank

We can generate three relaxed problems by removing one or both of the conditions:

(a) A tile can move from square A to square B if A is adjacent to B
(b) A tile can move from square A to square B if B is blank
(c) A tile can move from square A to square B

From (a) we can derive Manhattan distance. The reasoning is that h_2 would be the proper score if we moved each tile in turn to its destination. From (c) we can derive h_1 (misplaced tiles) because it would be the proper score if tiles could move to their intended destination in one step. If the relaxed problem is hard to solve, then the values of the corresponding heuristic will be expensive to obtain.

- **Absolver**

A program called ABSOLVER was written (1993) by A.E. Prieditis for automatically generating heuristics for a given problem. ABSOLVER generated a new heuristic for the 8-puzzle better than any pre-existing heuristic and found the first useful heuristic for solving the Rubik's Cube.

One problem with generating new heuristic functions is that one often fails to get one clearly bets heuristics. If a collection of admissible heuristics $h_1 \ldots\ldots h_m$ is available for a problem, and none of them dominates any of the others, then we need to make a choice. We can have the best of all worlds, by defining

$h(n) = \max\{h_1(n), \ldots\ldots\ldots\ldots\ldots\ldots h_m(n)\}$

This composite heuristics uses whichever function is most accurate on the node in the question. Because the component heuristics are admissible, h is admissible.

- **Sub-problems**

Admissible heuristics can also be derived from the solution cost of a sub problem of a given problem. Solution costs of sub-problems often serve as useful estimates of the overall solution cost. These are always admissible. For example, a heuristic for 8-puzzle might be the cost of moving tiles 1, 2, 3, 4 into their correct places. A common idea is to use a **pattern database** that stores the exact solution cost of every possible sub problem instance- in our example, every possible configuration of the four tiles and the blank.

Then we compute an admissible heuristics h_{DB} for each complete state encountered during a search simply by looking up the corresponding sub problem configuration in the database. The database itself is constructed by searching backwards from the goal state and recording the cost of each new pattern encountered. If multiple sub problems apply, take the maximum of the heuristics. If multiple disjoint sub problems apply, heuristics can be added.

- **Learn from experience**

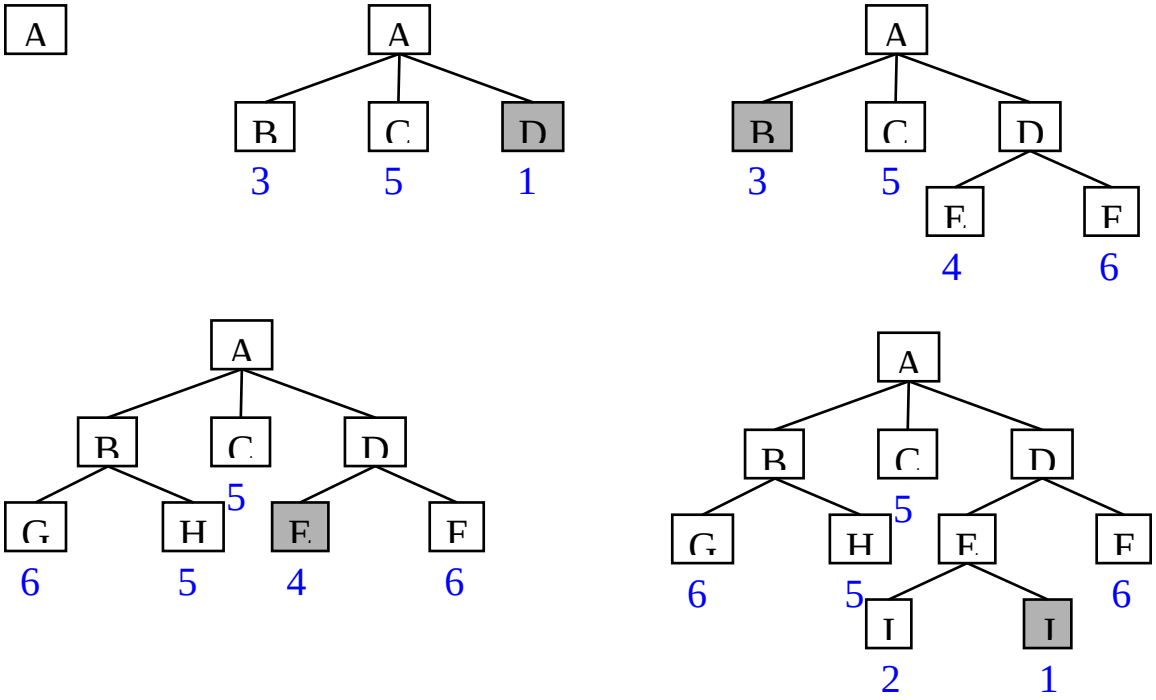
Another method to generate admissible heuristic functions is to learn from experience. Experience here means solving lots of 8-puzzles. Each optimal solution to an 8-puzzle problem provides examples from which h(n) can be learned. Learn from "features" of a state that are relevant to a solution, rather than just the raw state description (helps generalization). Generate "many" states with a given feature and determine average solution cost.

Combine information from multiple features using linear combination as:

h(n) = c1 * x1(n) + c2 * x2(n)… where x1, x2 are features and c1,c2 are constants.

Best First Search

Best First Search is a way of combining the advantages of both depth first search and breadth first search into a single method. Depth first search is good because it allows a solution to be found without all competing branches have to be expanded. Breadth first search is good because it does not get trapped on dead end paths. One way of combining two is **to follow a single path at a time, but switch paths whenever some competing path looks more promising than the current one does**.



At each step of the best first search process we select the most promising of the nodes we have generated so far. This is done by applying an **appropriate heuristic**

function to each of them. We then expand the chosen node by using the rules to generate its successors. If one of them is a solution then we can quit. If not all those new nodes are added to the set of nodes generated so far. Again the most promising node is selected and the process is repeated.

Usually what happens is that a bit of **depth first** searching occurs as the most promising branch is explored. But eventually if a solution is not found, that branch will start to look less promising than one of the top level branches that had been ignored. At that point the now more promising, previously ignored branch will be explored. But the old branch is not forgotten. Its last node remains in the set of generated but unexpanded nodes. The search can return to it whenever all the others get bad enough that it is again the most promising path.

Figure shows the beginning of a best first search procedure. Initially there is only one node, so it will be expanded. Doing so generates 3 new nodes. The heuristic function, which, in this example, is the cost of getting to a solution from a given node, is applied to each of these new nodes. Since node D is the most promising, it is expanded next, producing 2 successor nodes, E and F. But then the heuristic function is applied to them. Now another path, that going through node B, looks more promising, so it is pursued, generating nodes G and H. But again when these new nodes are evaluated they look less promising than another path, so attention is returned to the path through D to E. E is then expanded, yielding nodes I and J. At next step, J will be expanded since it is the most promising. This process can continue until a solution is found.

Although the example above illustrates a best first search of a tree, it is sometimes important to **search a graph** instead so that duplicate paths will not be pursued. An algorithm to do this will operate by searching a directed graph in which each node represents a point in the problem space. Each node will contain in addition to a description of the problem state it represents, an indication of how promising it is, a **parent link** that points back to the best node from which it came, and a list of the nodes that were generated from it.

The parent link will make it possible to recover the path to the goal once the goal is found. The list of successors will make it possible, if a better path is found to an already existing node, to propagate the improvement down to its successors. We will call a graph of this sort an **OR graph**, since each of its branches represents an alternative problem-solving path. To implement such a graph search procedure, we will need to use 2 lists of nodes:

- **OPEN**: nodes that have been **generated**, and have had the heuristic function applied to them but which have **not yet been examined** (i.e., had their successors generated). This is actually a priority queue in which the elements with the highest priority are those with the most promising values of the heuristic function.

- **CLOSED**: nodes that **have already been examined**. We need to keep these nodes in memory if we want to search a graph rather than a tree, since whenever a new node is generated; we need to check whether it has been generated before.

Algorithm:

1. Start with OPEN containing just the initial state.
2. Until a goal is found or there are no nodes left in OPEN do:
    a) Pick the best node in OPEN
    b) Generate its successors
    c) For each successor do:
        i. If it has not been generated before, evaluate it, add it to OPEN, and record its parent
        ii. If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.

Completeness: Yes. This means that, given unlimited time and memory, the algorithm will always find the goal state if the goal can possibly be found in the graph. Even if the heuristic function is highly inaccurate, the goal state will eventually be added to the open list and will be closed in some finite amount of time.

Time Complexity: It is largely dependent on the accuracy of the heuristic function. An inaccurate h does not guide the algorithm toward the goal quickly, increasing the time required to find the goal. For this reason, in the worst case, the Best-First Search runs in exponential time because it must expand many nodes at each level. This is expressed as $O(b^d)$, where b is the branching factor (i.e., the average number of nodes added to the open list at each level), and d is the maximum depth.

Space Complexity: The memory consumption of the Best-First Algorithm tends to be a bigger restriction than its time complexity. Like many graph-search algorithms, the Best-First Search rapidly increases the number of nodes that are stored in memory as the search moves deeper into the graph. One modification that can improve the memory consumption of the algorithm is to only store a node in the open list once, keeping only the best cost and predecessor. This reduces the number of nodes stored in memory but requires more time to search the open list when nodes are inserted into it. Even after this change, the space complexity of the Best-First Algorithm is exponential. This is stated as $O(b^d)$, where b is the branching factor, and d is the maximum depth.

Optimality: No. The Best-First Search Algorithm is not even guaranteed to find the shortest path from the start node to the goal node when the heuristic function perfectly estimates the remaining cost to reach the goal from each node. Therefore, the solutions found by this algorithm must be considered to be quick estimates of the optimal solutions.

A* Algorithm

A-Star (or A*) is a general search algorithm that is extremely competitive with other search algorithms, and yet intuitively easy to understand and simple to implement. Search algorithms are used in a wide variety of contexts, ranging from A.I. planning problems to English sentence parsing. Because of this, an effective search algorithm allows us to solve a large number of problems with greater ease.

The problems that A-Star is best used for are those that can be represented as a state space. Given a suitable problem, you represent the initial conditions of the problem with an appropriate initial state, and the goal conditions as the goal state. For each action that you can perform, generate successor states to represent the effects of the action. If you keep doing this and at some point one of the generated successor states is the goal state, then the path from the initial state to the goal state is the solution to your problem.

What A-Star does is generate and process the successor states in a certain way. Whenever it is looking for the next state to process, A-Star employs a heuristic function to try to pick the "best" state to process next. If the heuristic function is good, not only will A-Star find a solution quickly, but it can also find the best solution possible.

A search technique that finds minimal cost solutions and is also directed towards goal states is called "**A* (A-star) search**". A* algorithm is a typical heuristic search algorithm, in which the heuristic function is an estimated shortest distance from the initial state to the closest goal state, and it equals to traveled distance plus predicted distance ahead. In Best-First search and Hill Climbing, the estimate of the distance to the goal was used alone as the heuristic value of a state. In A*, we add the estimate of the remaining cost to the actual cost needed to get to the current state.

The A* search technique combines being "guided" by knowledge about where good solutions might be (the knowledge is incorporated into the Estimate function), while at the same time keeping track of how costly the whole solution is.

g(n) = the cost of getting from the initial node to n.
h(n) = the estimate, according to the heuristic function, of the cost of getting from n to the goal node.

That is, **f(n) = g(n) + h(n).** Intuitively, this is the estimate of the best solution that goes through n.

Like breadth-first search, A* is complete in the sense that it will always find a solution if there is one.If the heuristic function h is admissible, meaning that it never overestimates the actual minimal cost of reaching the goal, then A* is itself admissible (or optimal) if we do not use a closed set. If a closed set is used, then h must also be monotonic (or consistent) for A* to be optimal. This means that it never overestimates the cost of getting from a node to its neighbor. Formally, for all paths x,y where y is a successor of x:

Prepared By:

Er. Gurjot Singh Sodhi
Asst. Prof., Dept. of CSE
SUSCET, Tangori.

$$h(x) \leq g(y) - g(x) + h(y)$$

A* is also **optimally efficient** for any heuristic h, meaning that no algorithm employing the same heuristic will expand fewer nodes than A*, except when there are several partial solutions where h exactly predicts the cost of the optimal path.

A* is both admissible and considers fewer nodes than any other admissible search algorithm, because A* works from an "optimistic" estimate of the cost of a path through every node that it considers -- optimistic in that the true cost of a path through that node to the goal will be at least as great as the estimate. But, critically, as far as A* "knows", that optimistic estimate might be achievable.

When A* terminates its search, it has, by definition, found a path whose actual cost is lower than the estimated cost of any path through any open node. But since those estimates are optimistic, A* can safely ignore those nodes. In other words, A* will never overlook the possibility of a lower-cost path and so is admissible.

Suppose now that some other search algorithm A terminates its search with a path whose actual cost is not less than the estimated cost of a path through some open node. Algorithm A cannot rule out the possibility, based on the heuristic information it has, that a path through that node might have a lower cost. So while A might consider fewer nodes than A*, it cannot be admissible. Accordingly, A* considers the fewest nodes of any admissible search algorithm that uses a no more accurate heuristic estimate.

Algorithm:

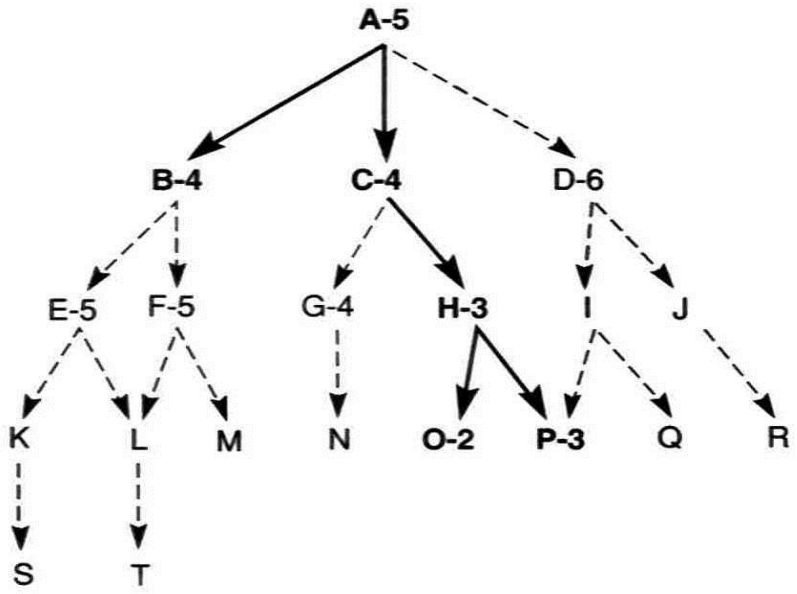
1. Create a search graph, G, consisting solely of the start node N1. Put N1 in a list called OPEN.
2. Create a list called CLOSED that is initially empty.
3. If OPEN is empty, exit with failure.
4. Select the first node on OPEN, remove it from OPEN, and put it on CLOSED. Call this node N.
5. If N is a goal node, exit successfully with the solution obtained by tracing a path along the pointers from N to N1 in G. (The pointers define a search tree and are established in step 7.)
6. Expand node N, generating the set, M, of its successors that are not already ancestors of N in G. Install these members of M as successors of N in G.
7. Establish a pointer to N from each of those members of M that were not already in G (i.e., not already on either OPEN or CLOSED). Add these members of M to OPEN. For each member, Mi, of M that was already on OPEN or CLOSED, redirect its pointer to N if the best path to Mi found so far is through N. For each member of M already on CLOSED, redirect the pointers of each of its descendants in G so that they point backward along the best paths found so far to these descendants.
8. Reorder the list OPEN in order of increasing *f* values. (Ties among minimal *f* values are resolved in favor of the deepest node in the search tree.)

Prepared By:

Er. Gurjot Singh Sodhi
Asst. Prof., Dept. of CSE
SUSCET, Tangori.

9. Go to step 3.

An example is shown below:



1. open = [A5]; closed = [ ]
2. evaluate A5; open = [B4,C4,D6]; closed = [A5]
3. evaluate B4; open = [C4,E5,F5,D6]; closed = [B4,A5]
4. evaluate C4; open = [H3,G4,E5,F5,D6]; closed = [C4,B4,A5]
5. evaluate H3; open = [O2,P3,G4,E5,F5,D6]; closed = [H3,C4,B4,A5]
6. evaluate O2; open = [P3,G4,E5,F5,D6]; closed = [O2,H3,C4,B4,A5]
7. evaluate P3; the solution is found!

Completeness: Yes, as long as branching factor is finite.

Time Complexity: It is largely dependent on the accuracy of the heuristic function. In the worst case, the A* Search runs in exponential time because it must expand many nodes at each level. This is expressed as $O(b^d)$, where b is the branching factor (i.e., the average number of nodes added to the open list at each level), and d is the maximum depth).

Space Complexity: The memory consumption of the A* Algorithm tends to be a bigger restriction than its time complexity. The space complexity of the A* Algorithm is also exponential since it must maintain and sort complete queue of unexplored options. This is stated as $O(b^d)$, where *b* is the branching factor, and *d* is the maximum depth.

Optimality: Yes, if *h* is admissible. However, a good heuristic can find optimal solutions for many problems in reasonable time.

Advantages:

1. A* benefits from the information contained in the Open and Closed lists to avoid repeating search effort.

2. A*'s Open List maintains the search frontier where as IDA*'s iterative deepening results in the search repeatedly visiting states as it reconstructs the frontier(leaf nodes) of the search.

3. A* maintains the frontier in sorted order, expanding nodes in a best-first manner.

Iterative-Deepening-A* (IDA*)

Iterative deepening depth-first search or **IDDFS** is a state space search strategy in which a depth-limited search is run repeatedly, increasing the depth limit with each iteration until it reaches *d*, the depth of the shallowest goal state. Analogous to this iterative deepening can also be used to improve the performance of the A* search algorithm. Since the major practical difficulty with A* is the large amount of memory it requires to maintain the search node lists, iterative deepening can be of considerable service. **IDA*** is a linear-space version of A*, using the same cost function.

Iterative Deepening A* is another combinatorial search algorithm, based on repeated depth-first searches. IDA* tries to find a solution to a given problem by doing a depth-first search up to a certain maximum depth. If the search fails, it is repeated with a higher search depth, until a solution is found. The search depth is initialized to a lower bound of the solution. Like branch-and-bound, IDA* uses pruning to avoid searching useless branches.

Algorithm:

1. Set THRESHOLD = heuristic evaluation of the start state
2. Conduct depth-first search, pruning any branch when its total cost exceeds THRESHOLD. If a solution path is found, then return it.
2. Increment THRESHOLD by the minimum amount it was exceeded and go to step 2.

Like A*, iterative deepening A* is guaranteed to find an optimal solution. Because of its depth first technique IDA* is very efficient with respect to space. IDA* was the first heuristic search algorithms to find optimal solution paths for the 15-puzzle (a 4x4 version of 8-puzzle) within reasonable time and space constraints.

Completeness: Yes, as long as branching factor is finite.

Time Complexity: Exponential in solution length i.e., $O(b^d)$, where *b* is the branching factor, and *d* is the maximum depth.

Space Complexity: Linear in the search depth i.e., O(d), where *d* is the maximum depth.

Optimality: Yes, if *h* is admissible. However, a good heuristic can find optimal solutions for many problems in reasonable time.

Advantages:

- Works well in domains with unit cost edges and few cycles
- IDA* was the first algorithm to solve random instances of the 15-puzzle optimally
- IDA* is simpler, and often faster than A*, due to less overhead per node
- IDA* uses a left-to-right traversal of the search frontier.

Disadvantages:

- May re-expand nodes excessively in domain with many cycles (e.g., sequence alignment)
- Real valued edge costs may cause the number of iterations to grow very large

Hill Climbing

Hill climbing is a **variant of generate and test** in which feedback from the test procedure is used to help the generator decide which direction to move in the search space. In a pure generate and test procedure the test function responds with only a "yes" or "no". But if the test function is augmented with a **heuristic function** that provides an

estimate of a how close a given state is to a goal state. This is particularly nice because often the computation of the heuristic function can be done at almost no cost at the same time that the test for a solution is being performed. Hill climbing is often used when a good heuristic function is available for evaluating states but when no other useful knowledge is available.

Hill climbing is an **optimization** technique which belongs to the family of Local search (optimization).Hill climbing attempts to **maximize** (or minimize) a **function** $f(x)$, where x are discrete states. These states are typically represented by vertices in a graph, where edges in the graph encode nearness or similarity of a graph. Hill climbing will follow the graph from vertex to vertex, always locally increasing (or decreasing) the value of f, until a **local maximum** x_m is reached.

1) Simple Hill Climbing

In hill climbing the basic idea is to always head towards a state which is better than the current one. So, if you are at town A and you can get to town B and town C (and your target is town D) then you should make a move IF town B or C appear nearer to town D than town A does.

Algorithm:

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise continue with initial state as the current state
2. Loop until a solution is found or there are no new operators left to be applied in the current state:
    a) Select an operator that has not yet been applied to the current state and apply it to produce a new state
    b) Evaluate the new state:
        i. If it is a goal state, then return it and quit.
        ii. If it is not a goal state but it is better than current state, then make it the current state
        iii. If it is not better than the current state, then continue in the loop

The key difference between this algorithm and the one we gave for generate and test is the use of an evaluation function as a way to inject task specific knowledge into the control process. For the algorithm to work, a precise definition of better must be provided. In some cases, it means a higher value of heuristic function. In others it means a lower value.

Note that the algorithm does not attempt to exhaustively try every node and path, so no node list or agenda is maintained - just the current state. If there are loops in the search space then using hill climbing you shouldn't encounter them - you can't keep going up and still get back to where you were before. Remember, from any given node, we go to the first node that is better than the current node. If there is no node better than the present node, then hill climbing halts.

2) Steepest-Ascent Hill Climbing

A useful variation on simple hill climbing considers all the moves from the current state and **selects the best one as the next state**. This method is called **Steepest-Ascent Hill Climbing or Gradient Search.** This is a case in which hill climbing can also operate on a continuous space: in that case, the algorithm is called **gradient ascent** (or gradient descent if the function is minimized).Steepest ascent hill climbing is similar to **best first search** but the latter tries all possible extensions of the current path in order whereas simple hill climbing only tries one. Following the steepest path might lead you to the goal faster, but then again, hill climbing does not guarantee you will find a goal.

Algorithm:

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise continue with initial state as the current state
2. Loop until a solution is found or until a complete iteration produces no change to the current state:

    a) Let SUCC be a state such that any possible successor of the current state will be better than SUCC
    b) For each operator that applies to the current state do:

        i. Apply the operator and generate the new state.
        ii. Evaluate the new state. If it is a goal state, then return it and quit. If it is not a goal state, compare it to SUCC. If it is better, then set SUCC to this state. If it is not better, leave SUCC alone.

    c) If the SUCC is better than the current state, then set the current state to SUCC

In **simple hill climbing**, the first closer node is chosen whereas in **steepest ascent hill climbing** all successors are compared and the closest to the solution is chosen. Both forms fail if there is no closer node. This may happen if there are local maxima in the search space which are not solutions. **Both** basic and steepest ascent hill climbing may **fail to find a solution**. Either algorithm may terminate not by finding a goal state but by getting to a state from which no better states can be generated. This will happen if the program has reached either a **local maxima, a ridge or a plateau**.

Problems

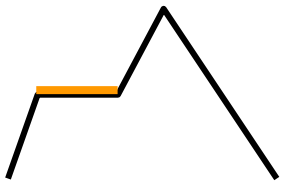
1) Local Maxima

A problem with hill climbing is that it will find only local maxima. **Local maximum** is a state that is better than all of its neighbours, but is not better than some

other states far away. At a local maximum, all moves appear to make things worse. Local maxima are particularly frustrating because they often occur within sight of a solution. In this case they are called **foothills**. Unless the heuristic is good / smooth, it need not reach a global maximum. Other local search algorithms try to overcome this problem such as stochastic hill climbing, random walks and simulated annealing. One way to solve this problem with standard hill climbing is to iterate the hill climbing.

2) Plateau

Another problem with Hill climbing is called the **Plateau** problem. This occurs when we get to a "flat" part of the search space, i.e. we have a path where the heuristics are all very close together and we end up wandering aimlessly. It is a **flat area** of search space in which a whole set of neighboring states have the **same value**. On a plateau it is not possible to determine the best direction in which to move by making local comparisons.



3) Ridges

A ridge is a curve in the search place that leads to a maximum. But the orientation of the ridge compared to the available moves that are used to climb is such that each move will lead to a smaller point. In other words to the algorithm each point on a ridge looks a like a local maximum even though the point is part of a curve leading to a better optimum. It is an area of the search space that is higher than surrounding areas and that itself has a slope. However, two moves executed serially may increase the height.

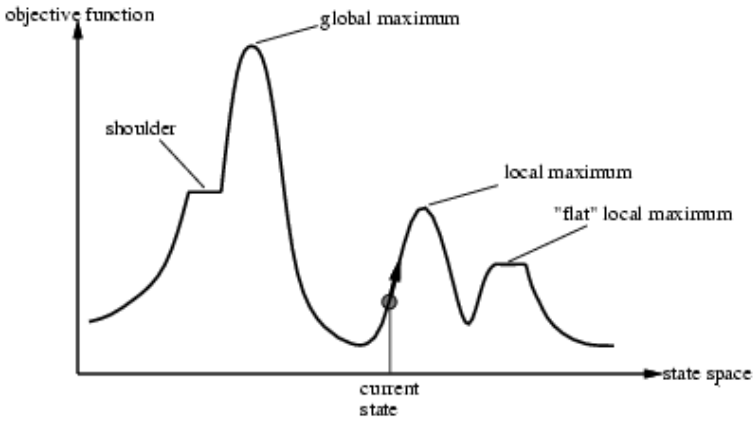


Figure above shows a one dimensional state space landscape in which elevation corresponds to the objective function or heuristic function. The aim is to find the global

maxima. Hill climbing search modifies the current state to try to improve it, as shown by the arrow.

Some ways of dealing with these problems are:

- **Backtrack** to some earlier node and try going in a different direction. This is particularly reasonable if at that node there was another direction that was looked as promising or almost as promising as the one that was chosen earlier. To implement this strategy, maintain a list of paths almost taken and go back to one of them if the path that was taken leads to a dead end. This is a fairly good way of dealing with local maxima.

- **Make a big jump in some direction** to try to get to a new section of the search space. This is particularly good way of dealing with plateaus. If the only rules available describe single small steps, apply them several times in the same direction.

- **Apply 2 or more rules before doing the test**. This corresponds to moving in several directions at once. This is particularly good strategy of dealing with ridges.

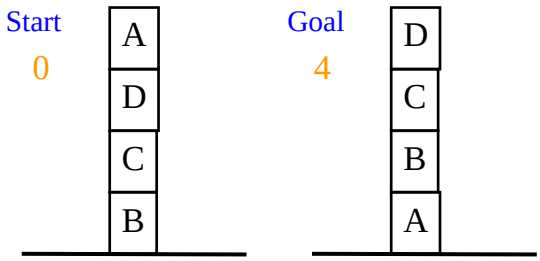
Even with these first aid measures, hill climbing is not always very effective. Hill climbing is a local method and it decides what to do next by looking only at the "immediate" consequences of its choices. Although it is true that hill climbing procedure itself looks only one move ahead and not any farther, that examination may in fact exploit an arbitrary amount of global information if that information is encoded in the heuristic function. Consider the blocks world problem shown in figure below. Assume the operators:

1. Pick up one block and put it on the table
2. Pick up one block and put it on another one

Suppose we use the following heuristic function:

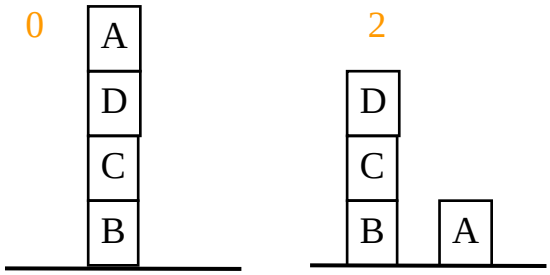
Local heuristic:

1. Add 1 point for every block that is resting on the thing it is supposed to be resting on.
2. Subtract 1 point for every block that is resting on a wrong thing.

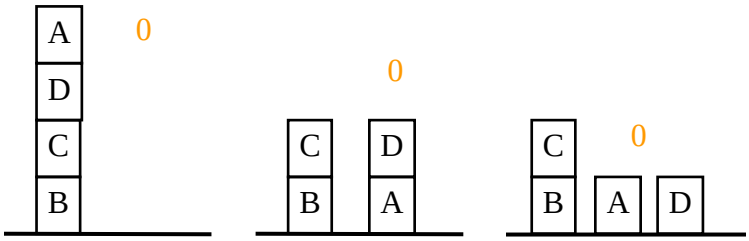


Blocks World

Using this function, the goal state has a score of 4. The initial state has a score of 0 (since it gets 1 point added for blocks C and D and one point subtracted for blocks A and B). There is only 1 move from initial state, namely to move block A to the table. That produces a state with a score of 2 (since now A's position causes a point to be added than subtracted). The hill climbing will accept that move.



From the new state there are 3 possible moves, leading to the 3 states shown in figure below. These states have scores: (a) 0, (b) 0, and (c) 0. Hill climbing will halt because all these states have lower scores than the current state. This process has reached a local maximum that is not the global maximum.

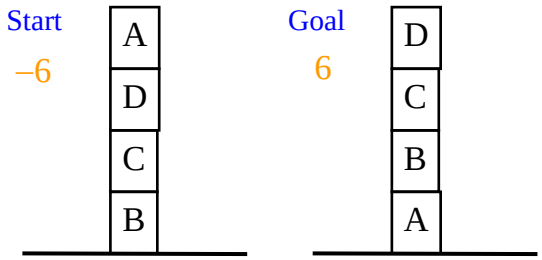


The problem is that by purely local examination of local structures, the current state appears to be better than any of its successors because more blocks rest on the correct objects. Suppose we try the following heuristic function in place of first one:

Global heuristic:

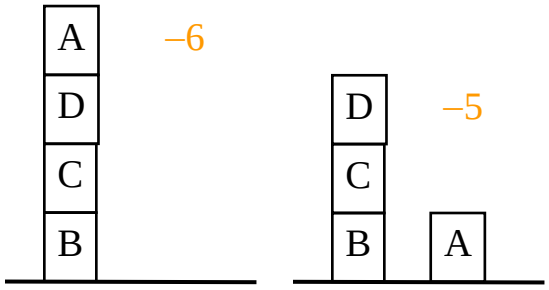
1. For each block that has the correct support structure (i.e. the complete structure underneath it is exactly as it should be), add 1 point to every block in the support structure.

2. For each block that has a wrong support structure, subtract 1 point for every block in the existing support structure.

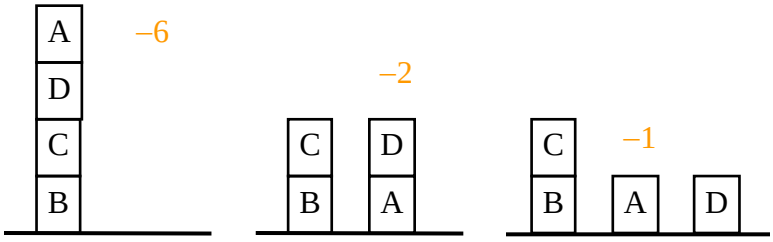


Blocks World

Using this heuristic function, the goal state has the score 6(1 for B, 2 for C, etc.). The initial state has the score -6. Moving A to the able yields a state with a score of -5 since A no longer has 3 wrong blocks under it.



The 3 new states that can be generated next now have the following scores: (a) -6,(b) -2, and (c) -1. This time steepest ascent hill climbing will chose move (c), which is the correct one.



This new heuristic function captures the 2 key aspects of this problem: incorrect structures are bad and should be taken apart; and correct structures are good and should be built up. As a result the same hill climbing procedure that failed with the earlier heuristic function now works perfectly. Unfortunately it is not always possible to construct such a perfect heuristic function. Often useful when combined with other methods, getting it started right in the right general neighbourhood.

Hill climbing is used widely in artificial intelligence fields, for reaching a goal state from a starting node. Choice of next node/ starting node can be varied to give a list of related algorithms. Some of them are:

1) **Random-restart hill climbing**

Random-restart hill climbing is a meta-algorithm built on top of the hill climbing algorithm. It is also known as **Shotgun hill climbing**. Random-restart hill climbing simply runs an outer loop over hill-climbing. Each step of the outer loop chooses a random initial condition x_0 to start hill climbing. The best x_m is kept: if a new run of hill climbing produces a better x_m than the stored state, it replaces the stored state.

It conducts a series of hill climbing searches from randomly generated initial states stopping when a goal is found. Random-restart hill climbing is a surprisingly effective algorithm in many cases. It turns out that it is often better to spend CPU time exploring the space, rather than carefully optimizing from an initial condition.

2) **Stochastic hill climbing**

Stochastic hill climbing chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than the steepest ascent, but in some state landscapes it finds better solutions.

3) **First-choice hill climbing**

First-choice hill climbing implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many (e.g., thousands) of successors.

Simulated Annealing

Simulated Annealing is a **variation of hill climbing** in which, at the beginning of the process, some **downhill** moves may be made. The idea is to do enough exploration of the whole space early on, so that the final solution is relatively insensitive to the starting state. This should lower the chances of getting caught at a local maximum, or plateau, or a ridge.

In order to be compatible with standard usage inn discussions of simulated annealing we make 2 notational changes for the duration of this section. We use the term **objective function** in place of the term heuristic function. And we attempt to **minimize** rather than maximize the value of the objective function.

Module 2

Simulated Annealing as a computational process is patterned after the physical process of **annealing** in which **physical substances** such as metals are **melted** (i.e. raised to high energy levels) and then **gradually cooled** until some solid state is reached. The goal of the process is to produce a **minimal-energy final** state. Physical substances usually move from higher energy configurations to lower ones, so the valley descending occurs naturally. But there is some **probability** that a **transition** to a **higher energy state** will **occur**. This probability is given by the function

$$p = e^{-\Delta E/kT}$$

where ΔE = positive change in energy level
T = temperature
k = Boltzmann's constant

Thus in the physical valley descending that occurs during annealing, the probability of a large uphill move is lower than the probability of a small one. Also, the probability that an uphill move will be made decreases as the temperature decreases. Thus such moves are more likely during the beginning of the processes when the temperature is high, and they become less likely at the end as the temperature becomes lower.

The rate at which the system is cooled is called **annealing schedule**. Physical annealing processes are very sensitive to the annealing schedule. If cooling occurs too rapidly, stable regions of high energy will form. In other words a local but not global minimum is reached. If however a slower schedule is used a uniform crystalline structure which corresponds to a global minimum is more likely to develop.

These properties of physical annealing can be used to define an analogous process of simulated annealing which can be used whenever simple hill climbing can be used. In this analogous process ΔE is generalized so that it represents not specifically the change in energy but more generally the change in the value of the **objective function**. The variable k describes the correspondence between the units of temperature and the units of energy. Since in this analogous process the units of both T and E are artificial it makes sense to incorporate k into T, selecting values for T that produce desirable behaviour on the part of the algorithm. Thus we used the revised probability formula

$$p^1 = e^{-\Delta E/T}$$

The algorithm for simulated annealing is only slightly different from the simple hill climbing procedure. The three differences are:

- The annealing schedule must be maintained
- Move to worse states may be accepted
- It is a god idea to maintain, in addition to the current state, the best state found so far. Then if the final state is worse than the earlier state, the earlier state is till available.

Prepared By:

Er. Gurjot Singh Sodhi
Asst. Prof., Dept. of CSE
SUSCET, Tangori.

Algorithm:

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise continue with initial state as the current state.
2. Initialize BEST-SO-FAR to the current state.
3. Initialize T according to the annealing schedule.
4. Loop until a solution is found or there are no new operators left to be applied in the current state:
    a) Select an operator that has not yet been applied to the current state and apply it to produce a new state.
    b) Evaluate the new state. Compute
    ΔE = Value of current state – Value of new state
        i. If the new state is a goal state, then return it and quit.
        ii. If it is not a goal state but it is better than the current state, then make it the current state. Also set BEST-SO-FAR to this new state.
        iii. If it is not better than the current state, then make it the current state with the probability p^1 as defined above. This step is usually implemented by invoking a random number generator to produce a number in the range [0,1]. If that number is less than p^1, then the move is accepted. Otherwise, do nothing.
    c) Revise T as necessary according to the annealing schedule
5. Return BEST-SO-FAR as the answer.

To implement this revised algorithm, it is necessary to select an annealing schedule which has 3 components.

1) The initial value to be used for temperature
2) The criteria that will be used to decide when the temperature of the system should be reduced
3) The amount by which the temperature will be reduced each time it is changed.

There may also be a fourth component of the schedule namely when to **quit**. Simulated annealing is often used to solve problems in which number of moves from a given state is very large. For such problems, it may not make sense to try all possible moves. Instead, it may be useful to exploit some criterion involving the number of moves that have been tried since an improvement was found.

While designing a schedule the first thing to notice is that as T approaches zero, the probability of accepting a move to a worse state goes to zero and simulated annealing becomes identical to hill climbing. The second thing to notice is that what really matters in computing the probability of accepting a move is the ratio $\Delta E/T$. Thus it is important that values of T be scaled so that this ratio is meaningful.

Prepared By:

Er. Gurjot Singh Sodhi
Asst. Prof., Dept. of CSE
SUSCET, Tangori.

Game Playing

Games provided a structured task in which it was very easy to measure success or failure. Games did not obviously require large amounts of knowledge, thought to be solvable by straightforward search.Games are good vehicles for research because they are well formalized, small, and self-contained. They are therefore easily programmed. Games can be good models of competitive situations, so principles discovered in game-playing programs may be applicable to practical problems.

Game playing has been a major topic of AI since the very beginning. Beside the attraction of the topic to people, it is also because its close relation to "intelligence", and its well-defined states and rules. The most common used AI technique in game is search. In other problem-solving activities, state change is solely caused by the action of the agent. However, in Multi-agent games, it also depends on the actions of other agents who usually have different goals.

A special situation that has been studied most is "two-person zero-sum game", where the two players have exactly opposite goals. (Not all competition are zero-sum!). Given sufficient time and space, usually an optimum solution can be obtained for the former by exhaustive search, though not for the latter. However, for most interesting games, such a solution is usually too inefficient to be practically used.

Game playing is always popular with humans. Chess and Go are two popular games. Claude Shannon and Alan Turing introduced the first chess programs in 1950. Chess was chosen due to its simplicity, concrete representations, and popularity. Chess programs were viewed as existence proof of a machine doing something thought to require intelligence.

Having an opponent at the game introduces **uncertainty**. In all game programs, one should deal with the **contingency problem**. It refers to the fact that every branch of the search tree deals with a possible contingency that may arise. Games are often too hard to solve. Uncertainty arises since there is no enough time to calculate the exact consequences of any move. Games have heavy penalties for inefficiency. **Pruning** helps us remove parts of the search tree that make no difference to the game while **heuristic evaluation functions** get us close to a true utility of a state without having to complete a full search.

Representation of games

The **games** studied by game theory are well-defined mathematical objects. A game consists of a set of players, a set of moves (or strategies) available to those players, and a specification of payoffs for each combination of strategies. There are two ways of representing games that are common in the literature.

Prepared By:

Er. Gurjot Singh Sodhi
Asst. Prof., Dept. of CSE
SUSCET, Tangori.

1. Normal form

The normal (or strategic form) game is usually represented by a matrix which shows the players, strategies, and payoffs (see the example below). More generally it can be represented by any function that associates a payoff for each player with every possible combination of actions. In the accompanying example there are two players; one chooses the row and the other chooses the column.

	Player 2 chooses *Left*	Player 2 chooses *Right*
Player 1 chooses *Up*	**4**, **3**	**–1**, **–1**
Player 1 chooses *Down*	**0**, **0**	**3**, **4**

Normal form or payoff matrix of a 2-player, 2-strategy game

Each player has two strategies, which are specified by the number of rows and the number of columns. The payoffs are provided in the interior. The first number is the payoff received by the row player (Player 1 in our example); the second is the payoff for the column player (Player 2 in our example). Suppose that Player 1 plays *Up* and that Player 2 plays *Left*. Then Player 1 gets a payoff of 4, and Player 2 gets 3.

When a game is presented in normal form, it is presumed that each player acts simultaneously or, at least, without knowing the actions of the other. If players have some information about the choices of other players, the game is usually presented in extensive form.

2. Extensive form

The extensive form can be used to formalize games with some important order. Games here are often presented as trees (as pictured to the left). Here each vertex (or node) represents a point of choice for a player. The player is specified by a number listed by the vertex. The lines out of the vertex represent a possible action for that player. The payoffs are specified at the bottom of the tree. Fig : An extensive form game

Prepared By:

Er. Gurjot Singh Sodhi
Asst. Prof., Dept. of CSE
SUSCET, Tangori.

In the game pictured here, there are two players. *Player 1* moves first and chooses either *F* or *U*. *Player 2* sees *Player 1*'s move and then chooses *A* or *R*. Suppose that *Player 1* chooses *U* and then *Player 2* chooses *A*, then *Player 1* gets 8 and *Player 2* gets 2.

The extensive form can also capture simultaneous-move games and games with incomplete information. Either a dotted line or circle is drawn around two different vertices to represent them as being part of the same information set (i.e., the players do not know at which point they are).

Types of games

1. Symmetric and asymmetric

A symmetric game is a game where the payoffs for playing a particular strategy depend only on the other strategies employed, not on who is playing them. If the identities of the players can be changed without changing the payoff to the strategies, then a game is symmetric. Many of the commonly studied 2×2 games are symmetric. The standard representations of **prisoner's dilemma**, and the **stag hunt** are all symmetric games.

	E	F
E	1, 2	0, 0
F	0, 0	1, 2

An asymmetric game

Most commonly studied asymmetric games are games where there are not identical strategy sets for both players. For instance, the **ultimatum** game and similarly the **dictator** game have different strategies for each player. It is possible, however, for a game to have identical strategies for both players, yet be asymmetric. For example, the game pictured to the right is asymmetric despite having identical strategy sets for both players.

Prepared By:

Er. Gurjot Singh Sodhi
Asst. Prof., Dept. of CSE
SUSCET, Tangori.

2. Zero sum and non-zero sum

Zero sum games are a special case of constant sum games, in which choices by players can neither increase nor decrease the available resources. In zero-sum games the total benefit to all players in the game, for every combination of strategies, always adds to zero (more informally, a player benefits only at the expense of others). **Poker** exemplifies a zero-sum game (ignoring the possibility of the house's cut), because one wins exactly the amount one's opponents lose. Other zero sum games include classical board games such as **go** and **chess**.

	A	B
A	–1, 1	3, –3
B	0, 0	–2, 2

A zero-sum game

Many games studied by game theorists are non-zero-sum games, because some outcomes have net results greater or less than zero. Informally, in non-zero-sum games, a gain by one player does not necessarily correspond with a loss by another.

Constant sum games correspond to activities like theft and gambling, but not to the fundamental economic situation in which there are potential gains from trade. It is possible to transform any game into a (possibly asymmetric) zero-sum game by adding an additional dummy player (often called "the board"), whose losses compensate the players' net winnings.

3. Simultaneous and sequential

Simultaneous games are games where both players move simultaneously, or if they do not move simultaneously, the later players are unaware of the earlier players' actions (making them *effectively* simultaneous). Sequential games (or dynamic games) are games where later players have some knowledge about earlier actions. This need not be perfect knowledge about every action of earlier players; it might be very little information.

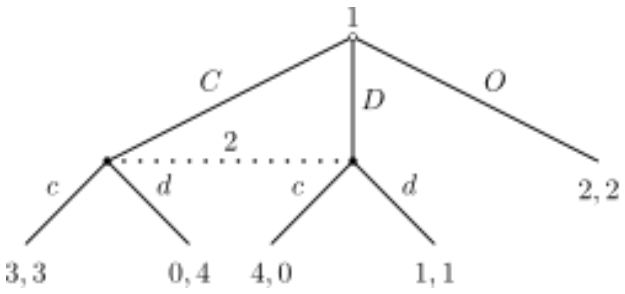
For instance, a player may know that an earlier player did not perform one particular action, while he does not know which of the other available actions the first player actually performed. Normal form is used to represent simultaneous games, and extensive form is used to represent sequential ones.

Prepared By:

Er. Gurjot Singh Sodhi
Asst. Prof., Dept. of CSE
SUSCET, Tangori.

4. Perfect information and imperfect information

A game of imperfect information (the dotted line represents ignorance on the part of player 2) is shown below. An important subset of sequential games consists of games of perfect information. A game is one of perfect information if all players know the moves previously made by all other players. Thus, only sequential games can be games of perfect information, since in simultaneous games not every player knows the actions of the others. Perfect information games include chess, go.



Perfect information is often confused with complete information, which is a similar concept. Complete information requires that every player knows the strategies and payoffs of the other players but not necessarily the actions.

5. Infinitely long games

Games, as studied by economists and real-world game players, are generally finished in a finite number of moves. Pure mathematicians are not so constrained, and set theorists in particular study games that last for infinitely many moves, with the winner (or other payoff) not known until *after* all those moves are completed.

The focus of attention is usually not so much on what is the best way to play such a game, but simply on whether one or the other player has a winning strategy. (It can be proven, using the axiom of choice, that there are games—even with perfect information, and where the only outcomes are "win" or "lose"—for which *neither* player has a winning strategy.) The existence of such strategies, for cleverly designed games, has important consequences in descriptive set theory.

Games as search problem

- **Initial state:** Initial State is the current board/position
- **Operators:** legal moves and their resulting states
- **A terminal test:** decide if the game has ended

Prepared By:

Er. Gurjot Singh Sodhi
Asst. Prof., Dept. of CSE
SUSCET, Tangori.

- **A utility function (Payoff function):** produces a numerical value for (only) the terminal states. Example: In chess, outcome = win/loss/draw, with values +1, -1, 0 respectively

Game Trees

The sequence of states formed by possible moves is called a **game tree** ; each level of the tree is called a **ply**. In 2 player games we call the two players **Max** (us) and **Min** (the opponent).WIN refers to winning for Max. At each ply, the "turn" switches to the other player.

Each level of search nodes in the tree corresponds to all the possible board configurations for a particular player – Max or Min. Utility values found at the end can be returned back to their parent nodes. So, winning for Min is *losing* for Max. Max wants to end in a board with +1 and Min in a board with a value of -1.

Max chooses the board with the max utility value, Min the minimum. Max is the first player and Min is the second. Every player needs a **strategy**. For example, the strategy for Max is to reach a winning terminal state regardless of what Min does. Even for simple games, the search tree is huge.

Minimax Algorithm

The **Minimax Game Tree** is used for programming computers to play games in which there are two players taking turns to play moves. Physically, it is just a tree of all possible moves.

With a full minimax tree, the computer could look ahead for each move to determine the best possible move. Of course, as you can see in example diagram shown below, the tree can get very big with only a few moves. Thus, for large games like Chess and Go, computer programs are forced to estimate who is winning or losing by focusing on just the top portion of the entire tree. In addition, programmers have come up with all sorts of **algorithms** and tricks such as Alpha-Beta pruning.

The minimax game tree, of course, cannot be used very well for games in which the computer cannot see the possible moves. So, minimax game trees are best used for games in which both players can see the entire game situation. These kinds of games, such as checkers, othello, chess, and go, are called **games of perfect information**.

For instance, take a look at the following (partial) search tree for Tic-Tac-Toe. Notice that unlike other trees like binary trees, 2-3 trees, and heap trees, a node in the game tree can have any number of children, depending on the game situation. Let us

assign points to the outcome of a game of Tic-Tac-Toe. If X wins, the game situation is given the point value of 1. If O wins, the game has a point value of -1. Now, X will be trying to **maximize** the point value, while O will be trying to **minimize** the point value. So, one of the first researchers on the minimax tree decided to name player X as Max and player O as Min. Thus, the entire data structure came to be called the **minimax game tree**.

The games we will consider here are:

- *two-person:* there are two players.
- *perfect information:* both players have complete information about the state of the game. (Chess has this property, but poker does not.)
- *zero-sum:* if we count a win as +1, a tie as 0, and a loss as -1, the sum of scores for both players is always zero.

Examples of such games are chess, checkers, and tic-tac-toe.

This minimax logic can also be extended to games like chess. In these more complicated games, however, the programs can only look at the part of the minimax tree; often, the programs can't even see the end of the game because it is so far down the tree. So, the computer only looks at a certain number of nodes and then stops. Then the computer tries to **estimate who is winning and losing** in each node, and these estimates result in a numerical point value for that game position. If the computer is playing as Max, the computer will try to maximize the point value of the position, with a win (checkmate) being equal to the largest possible value (positive 1 million, let's say). If the computer is playing as Min, it will obviously try to minimize the point value, with a win being equal to the smallest possible value (negative 1 million, for instance).

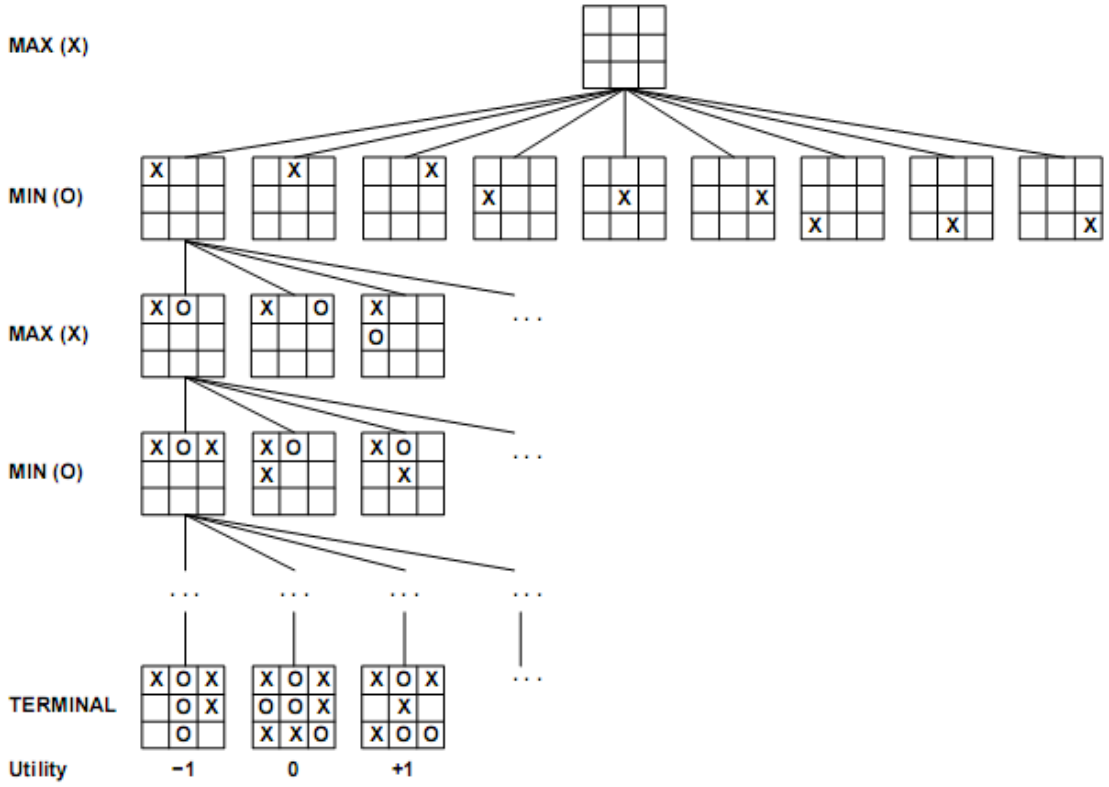


Fig : A (partial)search tree for the game of Tic-Tac-Toe is shown above. The top node is the initial state and max moves first placing an X in an empty square. We show part of the search tree, giving alternating moves by min(O) and max until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Minimax Evaluation

The Min-Max algorithm is applied in two player games, such as tic-tac-toe, checkers, chess, go, and so on. All these games have at least one thing in common, they are logic games. This means that they can be described by a set of rules and premises. With them, it is possible to know from a given point in the game, what are the next available moves. So they also share other characteristic, they are 'full information games'. Each player knows everything about the possible moves of the adversary.

- A search tree is generated, depth-first, starting with the current game position up to the end game position.
- Compute the values (through the utility function) for all the terminal states.
- Afterwards, compute the utility of the nodes one level higher up in the search tree (up from the terminal states. The nodes that belong to the MAX player receive the maximum value of its children. The nodes for the MIN player will select the minimum value of its children.

- Continue backing up the values from the leaf nodes towards the root.
- When the root is reached, Max chooses the move that leads to the highest value (optimal move).

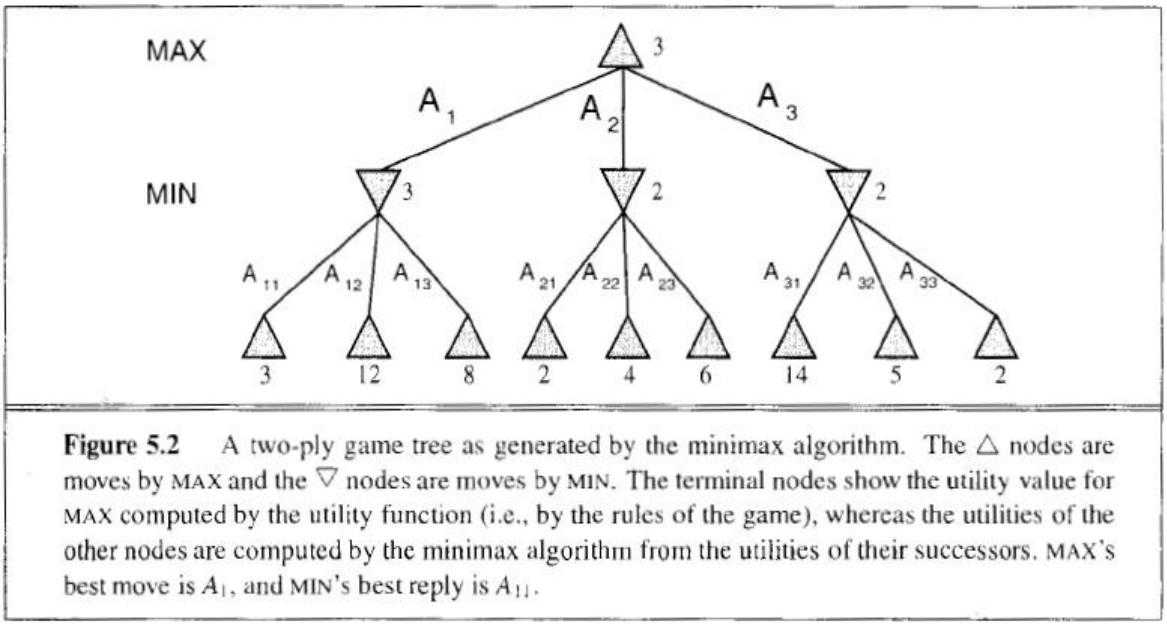
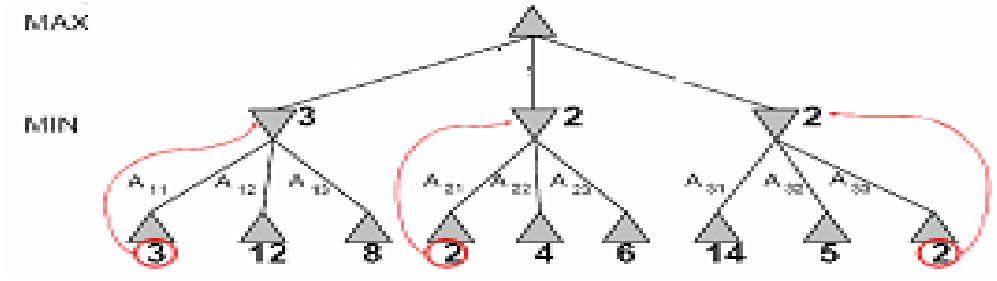


Figure 5.2 A two-ply game tree as generated by the minimax algorithm. The △ nodes are moves by MAX and the ▽ nodes are moves by MIN. The terminal nodes show the utility value for MAX computed by the utility function (i.e., by the rules of the game), whereas the utilities of the other nodes are computed by the minimax algorithm from the utilities of their successors. MAX's best move is A_1, and MIN's best reply is A_{11}.

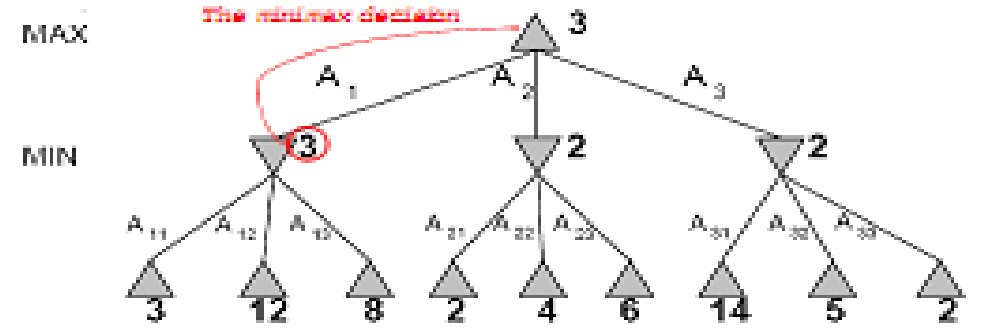
Given a game tree the optimal strategy can be determined by examining the minimax value of each node which we can write as MINIMAX-VALUE (n). Utility value is the value of a terminal node in the game tree. Minimax value of a terminal state is just its utility. Minimax value indicates the *best* value that the current player can possibly get. It's either the max or the min of a **bunch** of utility values. The minimax algorithm is a depth-first search. The space requirements are linear with respect to *b* and *m* where b is the no of legal moves at each point and m is the maximum depth of the tree. For real games, the time cost is impractical.

The minimax search will be as below:

Two-Ply Game Tree



Two-Ply Game Tree



Minimax maximizes the worst-case outcome for max.

Prepared By:

Er. Gurjot Singh Sodhi
Asst. Prof., Dept. of CSE
SUSCET, Tangori.

Algorithm:

```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

The MINIMAX value of a node can be calculated as

$$\text{MINIMAX-VALUE}(n) = \begin{cases} \text{UTILITY}(n) & \text{if } n \text{ is a terminal state} \\ \max_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if n is a MAX node} \\ \min_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if n is a MIN node} \end{cases}$$

Above shown is an algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The function MAXVALUE and MINVALUE go through the whole game tree all the way to the leaves to determine the backed up value of a state.

Completeness: Yes (if tree is finite)
Time complexity: $O(b^d)$, where d is the depth of the tree.
Space complexity: O(bd) (depth-first exploration), where d is the depth of the tree.
Optimality: Yes, provided perfect info (evaluation function) and opponent is optimal

Alpha-Beta Search

The problem with minimax search is that the no of game states it has to examine is exponential in the no of moves. Minimax helps us look ahead four or five ply in chess. Average human chess players can make plans six or eight ply ahead. But it is possible to compute the correct minimax decision without looking at every node in the game tree. **Alpha-beta pruning** can be used in this context**.** It is similar to the minimax algorithm (applied to the same tree) it returns the same move as minimax would but prunes branches that cannot possibly influence the final decision.

- *alpha* is the value of the best choice (highest value) we have found till now along the path for MAX.
- *beta* is the value of the best choice (lowest value) we have found so far along the path for MIN.

Alpha-beta pruning changes the values for *alpha* or *beta* as it moves and prunes a sub tree as soon as it finds out that it is worse than the current value for *alpha* or *beta.* If two nodes in the hierarchy have incompatible inequalities (no possible overlap), then we know that the node below will not be chosen, and we can stop search.

Implementing Alpha-Beta Search

Alpha-beta search is easily implemented by adding *&alpha* and *&beta* parameters to a depth-first minimax search. The alpha-beta search simply quits early and returns the current value when *&alpha* or *&beta* threshold is exceeded. Alpha-beta search performs best if the best moves (for Max) and worst moves (for Min) are considered first; in this case, the search complexity is reduced to $O(b^{d/2})$. For games with high symmetry (e.g. chess), a *transposition table* (Closed list) containing values for previously evaluated positions can greatly improve efficiency.

Alpha-Beta Search Example

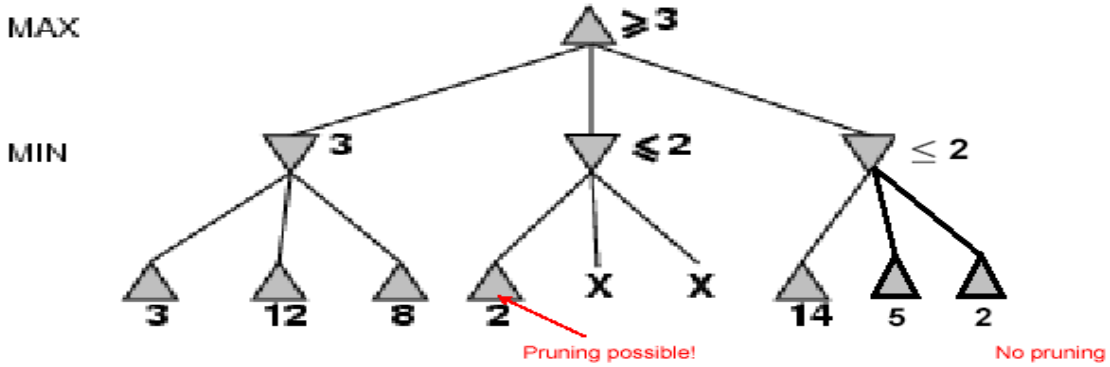
Bounded depth-first search is usually used, with the alpha-beta algorithm, for game trees. However:

1. The depth bound may stop search just as things get interesting (e.g. in the middle of a piece exchange in chess. For this reason, the depth bound is usually extended to the end of an exchange.
2. The search may tend to postpone bad news until after the depth bound: the *horizon effect*.

Frequently, large parts of the search space are irrelevant to the final decision and can be pruned. No need to explore options that are already definitely worse than the current best option. Consider again the 2 ply game tree from Fig 5.2. If we go through the calculation of optimal decision once more we can identify the minimax decision without ever evaluating 2 of the leaf nodes.

Prepared By:

Er. Gurjot Singh Sodhi
Asst. Prof., Dept. of CSE
SUSCET, Tangori.



Let the 2 unevaluated nodes be x and y and let z be the minimum of x and y. The value of root node is given by

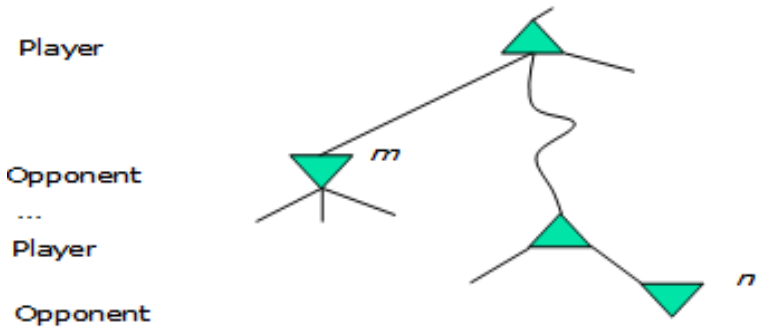
MINIMAX-VALUE (root) = max(min(3,12,8), min(2,x,y),min(14,5,2))

= max(3,min(2,x,y),2)

= max(3,z,2) where z≤2

= 3.

In other words the value of root and hence the minimax decision are independent of the values of the pruned leaves x and y. Alpha beta pruning can be applied to trees of any depth, and it is often possible to prune entire sub trees rather than just leaves. The general principle is this: consider a node n somewhere in the tree such that a player has a choice of moving to that node. If the player has a better choice m either at the parent node of n or at any choice point further up, then n will never be reached in actual play.



If *m* is better than *n* for Player, we will never reach *n* in play.

Prepared By:

Er. Gurjot Singh Sodhi
Asst. Prof., Dept. of CSE
SUSCET, Tangori.

Algorithm:

```
function ALPHA-BETA-SEARCH(state) returns an action
    inputs: state, current state in game
    v ← MAX-VALUE(state, −∞, +∞)
    return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
    inputs: state, current state in game
            α, the value of the best alternative for MAX along the path to state
            β, the value of the best alternative for MIN along the path to state
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s, α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v
    function MIN-VALUE(state, α, β) returns a utility value
        inputs: state, current state in game
                α, the value of the best alternative for MAX along the path to state
                β, the value of the best alternative for MIN along the path to state
        if TERMINAL-TEST(state) then return UTILITY(state)
        v ← +∞
        for a, s in SUCCESSORS(state) do
            v ← MIN(v, MAX-VALUE(s, α, β))
            if v ≤ α then return v
            β ← MIN(β, v)
        return v
```

The minimax search is depth first so at any one time we just have to consider the nodes along a single path in the tree. The effectiveness of alpha beta pruning is highly dependent on the order in which the successors are examined. In games repeated states occur frequently because of **transpositions**- different permutations of the move sequence that end up in the same position. It is worthwhile to store the evaluation of this position in a hash table the first time it is encountered so that we don't have to recompute it on subsequent occurrences.

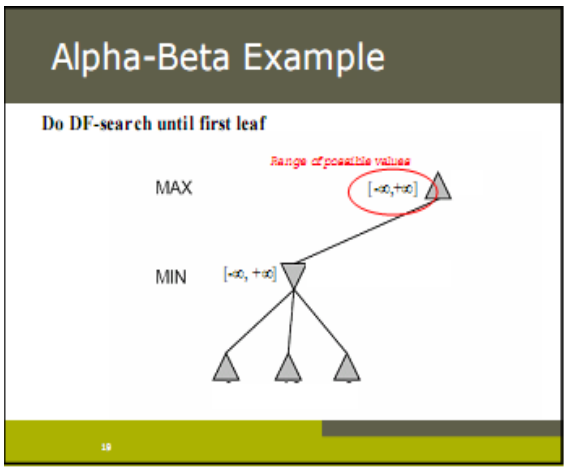
The hash table of previously seen positions is traditionally called transposition table. It is essentially identical to the CLOSED list in graph search. If we are evaluating a

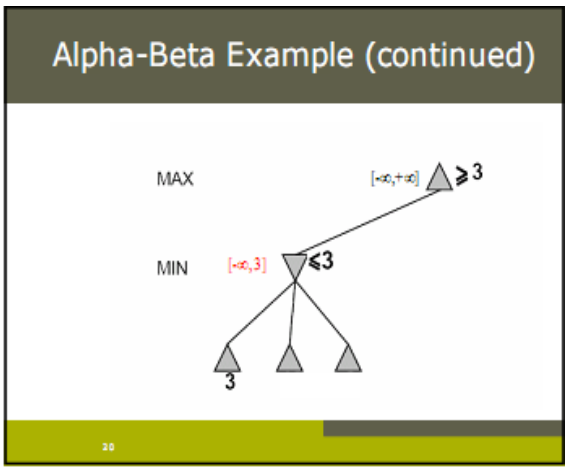
million nodes per second, it is not practical to keep all of them in the transposition table. Various strategies can be used to choose the most valuable ones.

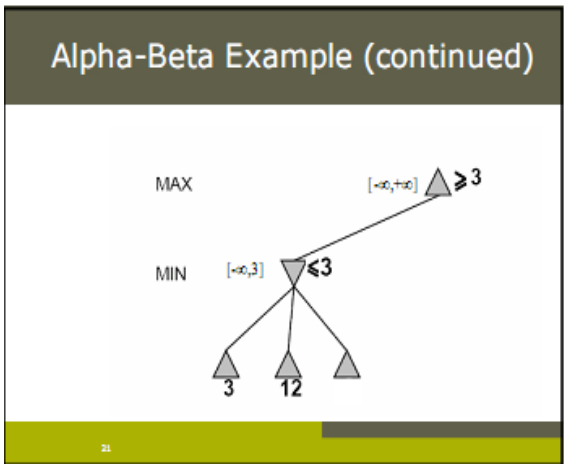
Limitations of alpha-beta Pruning

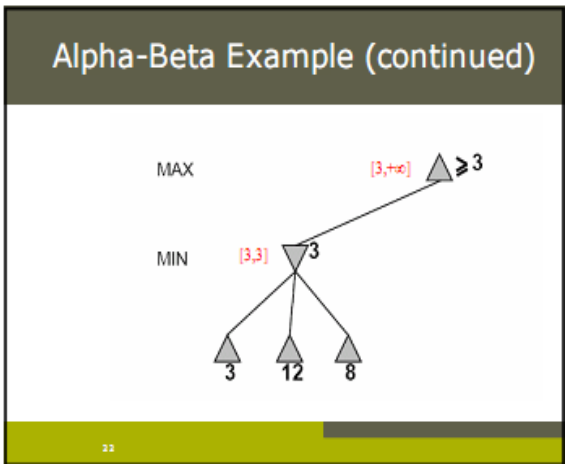
- Still dependant on search order. Can apply best-first search techniques
- Still has to search to terminal states for parts of the tree
- Depth may not be practical due to time Constraints

Completeness: Yes, provided the tree is finite
Time complexity: $O(b^d)$ in worst case where d is the depth of the tree. In best case Alpha-Beta, i.e., minimax with alpha-beta pruning, runs in time $O(b^{d/2})$, thus allowing us to evaluate a game tree twice as deep as we could with plain minimax.
Space complexity: O(bd) (depth-first exploration), where d is the depth of the tree
Optimality: yes, provided perfect info (evaluation function) and opponent is optimal
where d: Depth of tree and b: Legal moves at each point



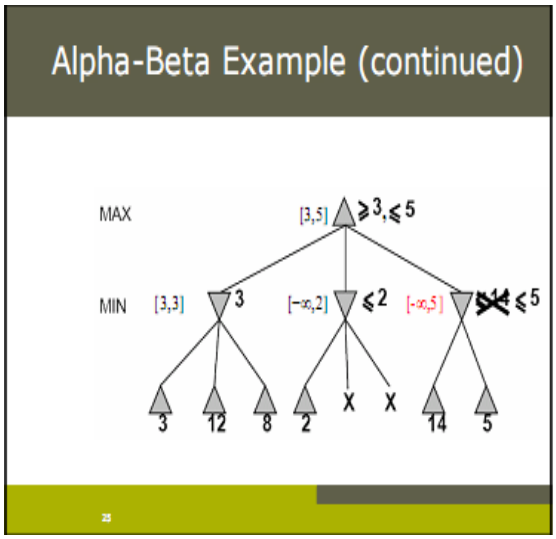


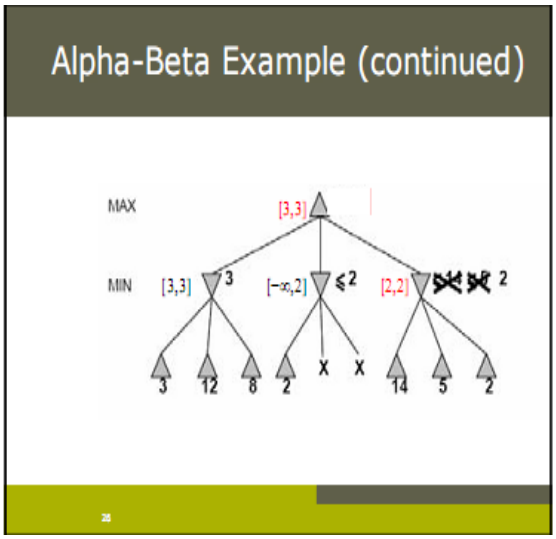


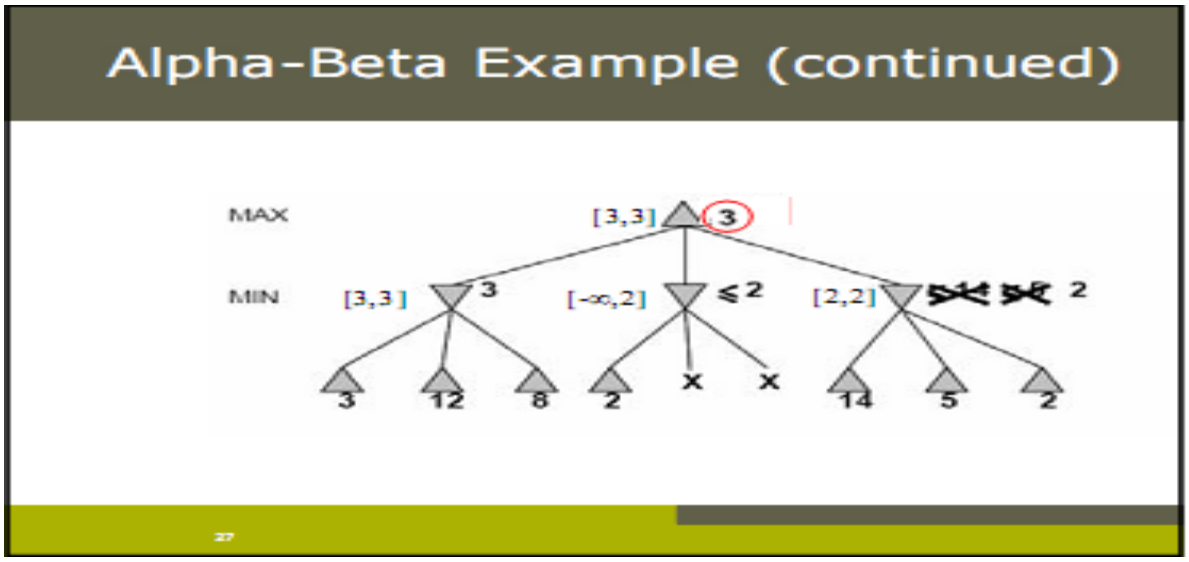


Prepared By:

Er. Gurjot Singh Sodhi
Asst. Prof., Dept. of CSE
SUSCET, Tangori.





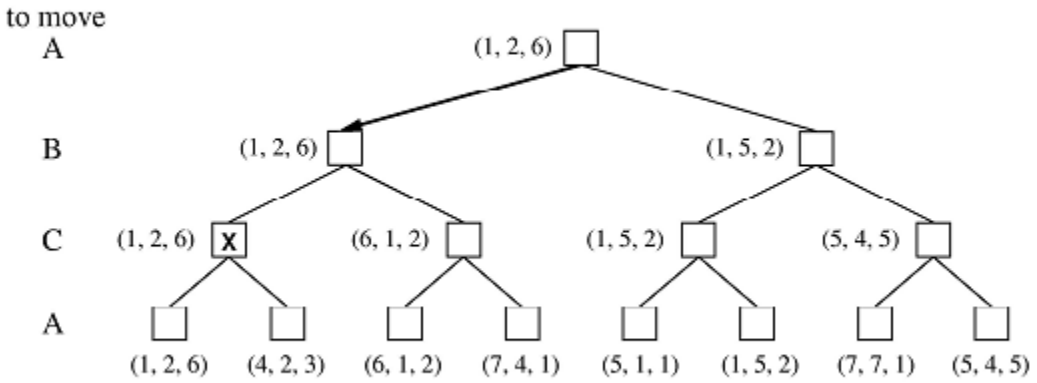


Optimal decisions in multiplayer games

Many popular games allow more than 2 players. Let us examine how to extend minimax idea to multiplayer games. First we need to replace the single values for each node with a vector of values. For e.g. in a 3 player game with players A, B and C a vector $<v_A,v_B,v_C>$ is associated with each node. For terminal state this vector gives the utility of the state from each player's viewpoint. In 2 player games the 2 element vector can be reduced to a single value because the values are always opposite. The simplest way to implement this is to have the utility function return a vector of utilities.

Prepared By:

Er. Gurjot Singh Sodhi
Asst. Prof., Dept. of CSE
SUSCET, Tangori.



Now we have to consider non terminal states. Consider the node marked X in the game tree shown below. In that state, player C chooses what to do. The two choices lead to terminal states with utility vectors $<v_A=1,v_B=2,v_C=6>$ and $<v_A=4,v_B=2,v_C=3>$. Since 6 is bigger than 3, C should choose the first move. This means that if state X is reached, subsequent play will lead to terminal states with utilities $<v_A=1,v_B=2,v_C=6>$. Hence the backed-up value of X is this vector. In general the backed-up value of a node n is the utility vector of whichever successor has the highest value for the player choosing at n.

Anyone who plays multiplayer games quickly becomes aware that there is a lot more going on than in 2 player games. Multiplayer games usually involve alliances, whether formal or informal, among the players. Alliances are made and broken as the game proceeds. In general the backed up value of a node is the utility vector of whichever successor has the highest value for the player choosing at n.

Imperfect Decisions

The minimax algorithm generates the entire game search space whereas the alpha beta algorithm allows us to prune large parts of it. However alpha beta search still has to search all the way to terminal states for at least a portion of the search space. The minimax is impractical since we assume that the program has time to search all the way to the terminal states. Shannon proposed the use of a heuristic evaluation function instead of the utility function that will enable us to stop the search earlier.

Although Alpha-Beta pruning helps to reduce the size of the search space, one still has to reach a terminal node before he can obtain a utility value. In other words, the suggestion is to alter minimax or alpha beta in 2 ways: the utility function is replaced by a heuristic evaluation function EVAL which gives us an estimate of the expected utility of the game from a given viewpoint and the terminal test is replaced by a cutoff test . For example, each pawn in chess is worth 1, a knight or bishop 3, and the queen 9.

Prepared By:

Er. Gurjot Singh Sodhi
Asst. Prof., Dept. of CSE
SUSCET, Tangori.

Evaluation function

An **evaluation function**, also known as a **heuristic evaluation function** or **static evaluation function**, is a function used by game-playing programs to estimate the value or goodness of a position in the minimax and related algorithms. The evaluation function is typically designed to be fast and accuracy is not a concern (therefore heuristic); the function looks only at the current position and does not explore possible moves (therefore static).

One popular strategy for constructing evaluation functions is as a weighted sum of various factors that are thought to influence the value of a position. For instance, an evaluation function for chess might take the form

c_1 * material + c_2 * mobility + c_3 * king safety + c_4 * center control + ...

Chess beginners, as well as the simplest of chess programs, evaluate the position taking only "material" into account, i.e they assign a numerical score for each piece (with pieces of opposite color having scores of opposite sign) and sum up the score over all the pieces on the board. On the whole, computer evaluation functions of even advanced programs tend to be more materialistic than human evaluations. This, together with their reliance on tactics at the expense of strategy, characterizes the style of play of computers.

Most evaluation functions work by calculating various **features** of a state for e.g. the number of pawns possessed by each side in the game of chess. The features taken together define various categories or equivalence classes of states: the states in each category have the same values for all the features. Any given category will contain some states that lead to wins, some that lead to draws and some that lead to losses. The evaluation function cannot know which states are which but it can return a single value that reflects the proportion of states with each outcome.

The evaluation function should not take long and should comply with the utility function on terminal states. Probabilities can help. For example, a position A with 40% chance of winning, 35% of losing, 25% of being a draw, has as evaluation:

1 x.4+-1x.35+0x.25=.05

In practice this kind of analysis requires too many categories and hence too much experience to estimate all the probabilities of winning. Instead most evaluation functions compute separate numerical contributions from each feature and then combine them to find the total value. Mathematically this kind of evaluation function is called a **weighted linear function** is

$$EVAL(s)=w_1f_1(s)+w_2f_2(s)+\ldots\ldots\ldots w_nf_n(s) = \sum_{i=1}^{n} w_if_i(s)$$

where w_i's are the weights (e.g., 1 for a pawn,5 for a rook,9 for a queen) and f_i 's are the features of a particular position. For chess the f_i could be the number of each kind of piece on the board and the w_i could be the values of the pieces.

Prepared By:

Er. Gurjot Singh Sodhi
Asst. Prof., Dept. of CSE
SUSCET, Tangori.

Adding up the values of features seems like a reasonable thing to do but in fact it involves a very strong assumption that the contribution of each feature is independent of the other features. A *static evaluation function* evaluates a position without doing any search.

Example 1: Tic-tac-toe

e(p) = nrows(Max) - nrows(Min) where nrows(i) is the number of complete rows, columns, or diagonals that are still open for player i.

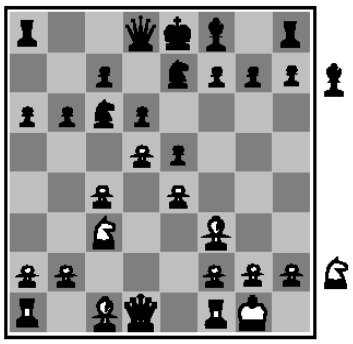
Example 2: Chess

For chess, typically *linear* weighted sum of features

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

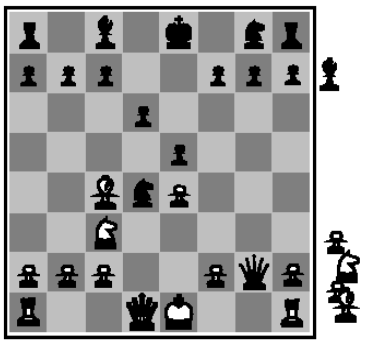
e.g., $w_1 = 9$ with
$f_1(s) =$ (number of white queens) – (number of black queens)
etc.

e(p) = (sum of values of Max's pieces) - (sum of values of Min's pieces) + k * (degree of control of the center)



Black to move

White slightly better



White to move

Black winning

Cutting off search

The next step is to modify alpha-beta search so that it will call heuristic EVAL function when it is appropriate to cut off the search.In terms of implementation we replace the 2 lines that mention TERMINAL-TEST with the following line.

If CUTOFF-TEST(state,depth) then return EVAL(state)

We must also arrange some bookkeeping so that the current depth is incremented on each recursive call. The most straight forward approach to controlling the amount of search is to set a fixed depth limit, so that CUTOFF-TEST returns true for all depth greater than some fixed depth d. Linear evaluation functions are used often but nonlinear ones are not uncommon. One should tune the weights by "trial-and-error". Another way is to apply iterative deepening. Heuristic functions evaluate board without knowing where *exactly* it will lead to. It is used to estimate the *probability* of winning from that node.

When searching a large game tree (for instance using minimax or alpha-beta pruning) it is often not feasible to search the entire tree, so the tree is only searched down to a certain depth. This results in the horizon effect where a significant change exists just over the "horizon" (slightly beyond the depth the tree has been searched). Evaluating the partial tree thus gives a misleading result. The **horizon effect** is more difficult to eliminate. It arises when the program is facing a move by opponent that causes serious damage and is ultimately unavoidable.

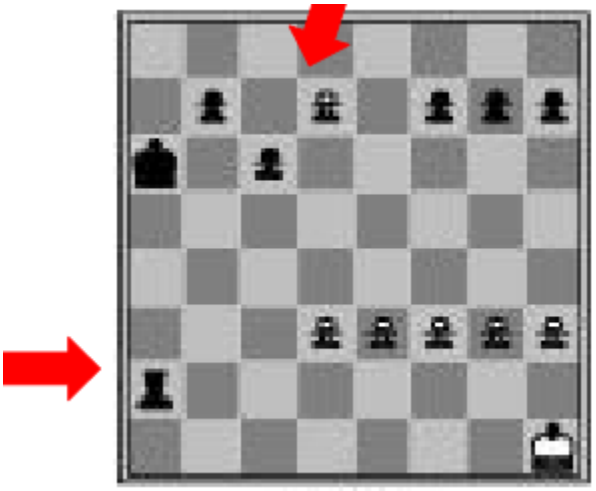
An example of the horizon effect occurs when some negative event is inevitable but postponable, but because only a partial game tree has been analyzed, it will appear to the system that the event can be avoided when in fact this is not the case. For example, in chess, if the white player is one move away from queening a pawn, the black AI can play moves to stall white and push the queening "over the horizon", and mistakenly believe that it avoided it entirely.

The horizon effect can be avoided by using "singular extension" in addition to the search algorithm. A singular extension is a move that is clearly better than all other moves in a given position. This gives the search algorithm the ability to look beyond its horizon, but only at moves of major importance (such as the queening in the previous example). This does not make the search much more expensive, as only a few specific moves are considered. In chess, the computer player may be looking ahead 20 moves. If

there are subtle flaws in its position that only matter after 40 moves, then the computer player can be beaten.

Horizon Effect

Fixed depth search thinks it can avoid the queening move



Black to move

The evaluation function should be applied only to positions which are **quiescent** (no dramatic changes in the near future). **Quiescence search** deals with the expansion of nonquiescent positions in order to reach quiescent positions. For example, one may try to postpone the queening move of a pawn "over the horizon" (where it cannot be detected). Essentially, a quiescent search is an evaluation function that takes into account some dynamic possibilities. Sometimes it is restricted to consider only certain types of moves such as capture moves that will quickly resolve the uncertainties in the position.

So far we have talked about cutting off search at a certain level and about doing alpha beta pruning that provably has no effect on the result. It is also possible to do forward pruning meaning that some moves at a given node are pruned immediately without further consideration. Clearly most humans playing chess only consider a few moves from each position. Unfortunately this approach is rather dangerous because there is no guarantee that the best move will not be pruned away. This can be disastrous if applied near the root, because every so often the program will miss some obvious moves. Combining all the techniques described here results in a program that can play creditable chess.

State-Of- The-Art-Game Programs

Introduction

We give some details of recent programs in variety of different types of games for playing two-person games. For some time there have been good programs for playing Backgammon, Othello and Draughts and in fact now there exist programs which are world champion status in all three areas. TD-Gammon has revolutionized the classic game of backgammon. TD-Gammon actually taught itself to play backgammon, starting from scratch, by using breakthrough research in Artificial Neural Networks. TD-Gammon learned to play well enough to rank among the best players in the world. Also Connect–4 and go–moku can be played perfectly by programs. We begin with Chess, where it has taken quite a while to produce a world–class program.

1) Chess

Deep Blue the chess program from IBM beat the world champion Kasparov 3.5 – 2.5 in May 1997. It has special purpose hardware and can evaluate 200 million moves per second. The special purpose hardware clearly helped, but carefully tuned software is also essential.

If we take a typical length game of chess, then there are 10125 possible nodes in the full game tree. This would take more than 10108 years to search if 109 moves could be investigated per second.

Deep Blue, uses search using mainly what is called singular extensions. It does say a 10 ply search, and then identifies the interesting lines of play to extend the search to a deeper level (say 30 – 40 ply).

It also has other types of extensions. Threat extension is used if a position of threat is discovered. Influence extension if for example there is a free pawn which can move to become a queen.

The software is used to perform the initial search and then the special purpose hardware for the last 5 ply. The evaluation heuristic function makes use of a great deal of information such as:

- how good a position is it e.g. difference in number of pieces;
- position of important pieces (e.g. bishop on black is no good if all opponents pieces on white);

Deep Blue is compared with grand masters games to tune the evaluation function. Also did some manual tuning by playing against a grand master until poor move identified. Much of the tree is evaluated in parallel using the 512 special purpose chips with minimal communication. IBM are interested in exploiting what has been learnt from Deep Blue in real world problems. These include such areas as:

- financial risk (using Monte Carlo simulations);
- data mining;

Prepared By:

Er. Gurjot Singh Sodhi
Asst. Prof., Dept. of CSE
SUSCET, Tangori.

• molecular dynamics simulation – of interest in pharmaceutical modelling (hope to deal with 100,000 atoms.
• use the visualisation tools developed for use in other domains

More recently, a program called X3D Fritz has come on the Chess scene and is a competitor to Deep Blue. X3D Fritz has played four games against Kasparov in November 2003. The scores were 1 win for Kasparov, 1 win for X3D Fritz and two draws. The game is played on a computer with a 3D chess board when viewed through special glasses. Kasparov received $175K for the games.

2) Checkers

The game of Checkers or Draughts is played on a board of sixty-four squares of alternate colors, usually referred as black and white although not necessarily of those colors. Twenty-four pieces (fat round counters), called men (twelve on each side) also of opposite colors are located on the game board. The game is played by two persons; the one having the twelve black or red pieces is technically said to be playing the tint side, and the other having the twelve white "men", to be playing the second side.

Samuels learning program eventually beat developer. Chinook is world champion, previous champ held title for 40 years had to withdraw for health reasons. It has all end games from 8 pieces pre–computed. This is 440×109 positions and is compressed to 6 gigabytes. It uses 25 play look-ahead tree, and so in the tree soon comes to end positions. Chinook drew with the world champion Tinsley in 1994. He was considered the best champion there has ever been in draughts.

The techniques developed are being used in BioTools which is a program for DNA synthesis. David Fogel has used evolutionary programming to automatically generate a draughts playing program called Blondie24. This was played against human players on the web and eventually obtained a rating of 2,000. The evolutionary program generated a neural network which was used to evaluate a heuristic function for mini-max.

The object of the game is to capture all of the opponent's men, or block them so they cannot be moved, the person whose side is brought to this state loses the game.

3) Othello

Reversi and Othello are names for a strategic board game which involves play by two parties on an eight-by-eight square grid with pieces that have two distinct sides. Pieces typically appear coin-like, but with a light and a dark face, each side representing one player. The object of the game is to make your pieces constitute a majority of the pieces on the board at the end of the game, by turning over as many of your opponent's pieces as you can.

Othello also called Reversi is probably more popular as a computer game than as a board game. It has a smaller search space than chess, usually 5 to 15 legal moves, but evaluation expertise had to be developed from scratch. In 1997 the Logistello program defeated the human world champion Takesh Murakami by 6 games to none. It is generally acknowledged that humans are no match for computers at Othello.

4) Backgammon

Most work on blackgammon has gone into improving the evaluation function. Gerry Tesauro combined Samuel's reinforcement learning method with neural network techniques to develop a remarkably accurate evaluator that is used with a search to depth 2 or 3. After playing more than a million training games against itself, Tesauro's program, TD Gammon is reliably ranked among the top 3 players in the world.

5) Go

Go is a strategic board game for two players. It is also known as **Weiqi** in Chinese, **Igo** or **Go** in Japanese. Go originated in ancient China, centuries before it was first mentioned in writing 548 BC. It is now popular throughout the world, especially in East Asia.

Go is played by two players alternately placing black and white stones on the vacant intersections of a 19×19 line grid. A stone or a group of stones is captured and removed if it is tightly surrounded by stones of the opposing color. The objective is to control a larger territory than the opponent's by placing one's stones so they cannot be captured. The game ends and the score is counted when both players consecutively pass on a turn, indicating that neither side can increase its territory or reduce its opponent's.

In game theory terms, Go is a zero-sum, perfect information, deterministic strategy game, putting it in the same class as chess, checkers (draughts), and reversi (othello), although it is not similar in its play to these. Although the game rules are very simple, the practical strategy is extremely complex.

The game emphasizes the importance of balance on multiple levels, and has internal tensions. To secure an area of the board, it is good to play moves close together; but to cover the largest area one needs to spread out, perhaps leaving weaknesses that can be exploited. Playing too low (close to the edge) secures insufficient territory and influence; yet playing too high (far from the edge) allows the opponent to invade. Many people find Go attractive for its reflection of the conflicting demands of real life.

6) Bridge

Contract bridge, usually known simply as bridge, is a trick-taking card game of skill and chance (the relative proportions depend on the variant played). It is played by four players who form two partnerships (sides); the partners sit opposite each other at a table. The game consists of the auction (often called bidding) and play, after which the hand is scored.

The bidding ends with a contract, which is a declaration by one partnership that their side shall take at least a stated number of tricks, with specified suit as trump or without trumps. The rules of play are similar to other trick-taking games with the addition of the fact that one player's hand is displayed face up on the table as the "dummy".

Much of bridge's popularity owes to the possibility that it can be played in tournaments of theoretically unlimited number of players; this form is referred to as duplicate bridge.